Anna Prochowska, 360709
Computational complexity, Homework 2

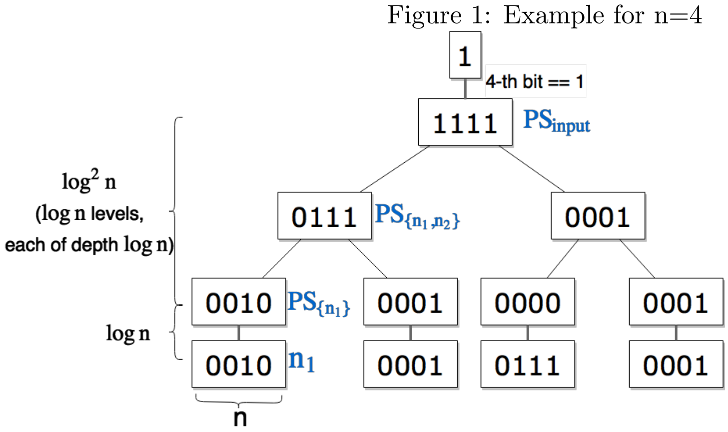
Problem 2.1

Define PS_S as $\{k \mid \exists_{S_i \subseteq S} \sum S_i = k\}$ (set of all possible sums of subsets of S). PS_S can be encoded as a binary number of length $max\ PS_S$: k-th bit is set iff $k \in PS_S$. In our solution PS_S is represented as a binary number of length n.

We describe a construction of a uniform circuit of depth $O(log^2\ n)$, polynomial size, and gates having fan-in 2. This proves that the problem is in $u - NC$.

Algorithm overview:
$input = \{n_1, ..., n_n\}$ is a set of n input numbers.
1. Convert each input number n_i to $PS_{\{n_i\}}$. Depth: $O(log\ n)$.
2. Compute PS_{input} in "divide and conquer" manner. To get PS_S, first split S into two subsets S_1, S_2 of (almost) equal size, compute PS_{S1} and PS_{S2}, and then use them to calculate PS_S. Depth: $O(log^2\ n)$.
3. Return 1 iff n-th bit of PS_{input} is set. This is equivalent to saying whether $n \in PS_{input}$. Depth: $O(1)$, because it is a single *and* operation of 1 and n-th bit of PS_{input}.

Figure 1: Example for n=4



Below are details of algorithm steps:
Ad. 1
The only possible sum of non-empty subset of $\{n_i\}$ is n_i, so at most one bit of $PS_{\{n_i\}}$ is set.
Our goal is to find a subset of numbers which sum up to n. Numbers are positive, therefore no number greater than n can be included in this set, so we can just skip numbers greater than n in our reasoning. So, if $n_i > n$ then $PS_{\{n_i\}} = 0$, otherwise exactly one bit of $PS_{\{n_i\}}$ is set, n_i-th one.

In other words, each input number n_i is converted to $f(n_i)$, where:

$$f(i) = \begin{cases} 2^{i-1} & i \leq n \\ 0 & otherwise \end{cases}$$

Below is a construction of a circuit of depth $O(log\ n)$ performing this operation.

For each of n output bits we create an independent circuit of depth $O(log\ n)$ and size $O(n)$ which returns 1 for i-th bit iff input is equal to i. Consider binary representation of i now. Bits at some positions are set $s_1, ..., s_k$, while others are unset $u_1, ..., u_l$, where $k + l = n$. Therefore value of i-th bit is equal to a value of bitwise operation s_1 & ... & s_k & $\neg u_1$ & ... & $\neg u_l$ There are $n - 1$ *and* operations which can be structured as a binary tree of depth $O(log\ n)$.

Ad. 2
To compute PS_S we first split S into two subsets S_1, S_2 of (almost) equal size ($|S1| = |S2|$ or $|S1| + 1 = |S2|$, $S1 \cup S2 = S$ and $S1 \cap S2 = \emptyset$). Then compute PS_{S1} and PS_{S2} recursively and finally use PS_{S1} and PS_{S2} to calculate PS_S. Operations of computing PS can be represented as a binary tree: node PS_S has children PS_{S1} and PS_{S2}. Tree root is PS_{input} while leaves are $PS_{\{n_i\}}$ (PS for singletons containing input numbers n_i). At i-th tree level, subsets of size at most $n/2^i$ are considered, because in each node set is divided into two subsets of (almost) equal size. Therefore tree depth is $log\ n$.

Now we explain how to compute PS_S out of PS_{S1} and PS_{S2}. This circuit is again of logarithmic depth.

$i \in PS_S$ iff we can select some numbers from $S1$ and $S2$ which sum up to i. Formally: there exist numbers a and b such that $a + b = i$, $a \in PS_{S1}$ and $b \in PS_{S2}$ (we assume that always $0 \in PS$). To verify if such a and b exist, we simply check all $i + 1$ possibilities: $i = 0 + i = 1 + (i - 1) = 2 + (i - 2) = ... = i + 0$. So i-th bit of PS_S is equal to $c_i = b_i \mid (a_1 \ \& \ b_{i-1}) \mid ... \mid a_i$ where $a_1 a_2 ...$ and $b_1 b_2 ...$ are PS_{S1} and PS_{S2}, respectively. Again, in order to use only logic gates with fan-in 2, so we must structure expression c_i as a binary tree. For each of n output bits we create a separate circuit. For each of them expression length is $O(n)$, so circuit depth is $O(log\ n)$.

To sum up, tree of PS operations has logarithmic depth and so has circuit computing PS in each node. Therefore, step 2 can be performed in depth $O(log^2\ n)$.

To show that described sequence of circuits is uniform, we must show that there exists a Turing Machine working in logarithmic space which on input 1^n outputs the representation of circuit C_n. Note that it's enough for a machine to store number n in binary, and counter from 1 to n in binary to create expression c_i or all i from i to n in stage 1. All the other operations can be encoded in states, so final circuit is uniform.

To sum up, described circuit has depth $O(log^2\ n)$. Size of input is n^2, so this variant of subset sum problem belongs to NC^2 complexity class, so in particular to NC.

Problem 2.2

To prove that problem in NP-complete we must show that, firstly, it belongs to NP, and secondly, it is NP-hard.

2) To prove that given problem is NP-hard, we reduce NP-complete problem 3-SAT in polynomial time to it.
Consider an instance of 3-SAT problem consisting of n clauses and k variables. Now we show how to convert it to a context-free grammar $G = (V, \Sigma, R, S)$.

V consists of a start symbol S, k nonterminals X_i and n nonterminals Y_i.
Σ contains n terminals a_i, each of them corresponds to one nonterminal Y_i.
Production rules are described as follows:
$\forall_{i \in \{1..n\}}$ there is a production rule $Y_i \to a_i$.
Also, for each variable x_i we have a corresponding non-terminal X_i. For each non-terminal X_i there are two production rules $X_i - > ....$ Let's call them *positive* and *negative* to distinguish between them.
Now consider i-th clause of the formula. Now we will define rules which contain symbol Y_i on their right hand side. The clause consists of 3 (positive or negative) literals. Consider one of them $(\neg)x_j$. If it is negative then symbol Y_i is included in a *negative* rule of nonterminal X_j, otherwise symbol Y_i is included in a *positive* rule of nonterminal X_j (right hand side of a rule is a concatenation of some nonterminals as described above; their order in a rule doesn't matter). Intuitively, we interpret i-th clause as a condition if nonterminal Y_i is used in a derivation: "nonterminal Y_i is used iff at least one the rules which contain it on the right hand side is used in a derivation tree".
For a start symbol S we have one rule: $Rule_S = S \to X_1 X_2 ... X_k$.

Example. Consider a formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_3)$. Production rules:
$S \to X_1 X_2 X_3$
$X_1 \to Y_1$
$X_2 \to Y_1 \mid Y_2$
$X_3 \to Y_2 \mid Y_1 Y_2$
$Y_1 \to a_1$
$Y_2 \to a_2$

Note that above algorithm of transformation from 3-SAT formula to a CFG works in time linear to size of input formula.

Lemma 1. Assigning values to variables in a 3-SAT boolean formula determines derivation tree in described CFG and vice versa.

Proof. Assigning a value to a variable x_i is an equivalent operation to choosing exactly one of *positive* and *negative* rules for nonterminal X_i. Number of variables k is the same as number of nonterminals X_i. There is 1:1 mapping between variables and X_i nonterminals. Also, for a variable we choose exactly one of two values 0 and 1. For a nonterminal, we also select one of two rules and we do it exactly once in every derivation tree. It must appear in a derivation tree as a part of rule $Rule_S$, but it doesn't appear on right hand side of any other rule.
For all the other nonterminals (S, Y_1, ..., Y_n) there is exactly one rule to choose from. So assigning values to all x_i variables determines a derivation tree and vice versa, for each derivation tree there is exactly one equivalent assignment. □

We claim that 3-SAT formula is satisfiable iff there exists a derivation in described CFG which uses all non-terminals.
$\Rightarrow$
Assume that formula is satisfiable so there exists an assignment for which all clauses are true.
Now we will show that it implies that all nonterminals Y_i are used in a corresponding derivation (which exists by Lemma 1). Consider i-th clause. Its 3 literals describe rules which contain Y_i, by definition of this grammar. At least one of the literals is true, so a rule containing Y_i on its right hand side belongs to a derivation. Therefore Y_i is used in a derivation. This reasoning applies to all Y_i nonterminals.
All derivations use symbol S and all the X_i nonterminals, because they are included in the only rule for start symbol S.
So all non terminals are used.
$\Leftarrow$
Now we will prove that following the statement holds: if Y_i is used in a derivation then i-th clause is true.
Each nonterminal Y_i appears on the right hand side of at most 3 rules which follows from a construction of a grammar. (It can appear less than 3 times if a particular literal is repeated in a clause, e.g. $x_1 \vee x_1 \vee x_1$). The fact that Y_i is used implies that at least one of the mentioned (at most 3) rules is used in a derivation. Choosing a rule $X_j \rightarrow ...$ corresponds to assigning a value to a variable x_j by Lemma 1. Assume that rule described by literal $(\neg)x_j$ is selected. If literal is positive, then we choose a *positive* rule and value 1 is assigned to x_j, so i-th clause is true. Otherwise literal is negative, then we select a *negative* rule and value 0 is assigned to x_j, so i-th clause is true in this case too.
All nonterminals Y_i are used so all clauses are true.

1) To prove that problem is in NP we show that there exists a certificate we can verify in polynomial time.
Consider a certificate of polynomial size which is a part of a derivation which contains all the nonterminals. (Below we will show that such certificate exists.) Once we have a derivation we should verify if it's correct. To do that, we first check if it's possible to apply rules as defined by this derivation. This simulation is linear to the size of the certificate.
Note that our certificate is not a full derivation tree, but only a part of it, so we have to check one more thing to make sure it's correct: all the nonterminals which are leaves of the certificate should have a finite derivation (can't loop). We can generate a set S of nonterminals which don't loop in the following way:
1. $S = \{N \mid N$ is nonterminal; there exists a production rule $N \to a_1 ... a_n$ where a_i are terminals$\}$
2. Add nonterminal M to S if there exists a rule $M \to ...N...$ for some $N \in S$.
3. Repeat step 2 until no more nonterminals can be added to S.
This algorithm is linear with regards to grammar size, because each nonterminal can be added to S at most once and total number of rules we iterate through is O(total size of the rules).

Now we prove following statement: if there exists a derivation which uses all nonterminals then there exists a part of this derivation tree of polynomial size which uses all nonterminals.
Let n be the number of nonterminals and m be a number of production rules.
Note that part of the tree we need has at most n leaves, because otherwise there would be a redundant nonterminal in a leaf. Now consider a path from the root to one of the leaves. If this path is longer than $m \times n$ then there exists a cycle which we can remove it and still have a valid certificate. If there is no cycle on a path, then for each nonterminal we can apply each rule at most once.

Therefore, if we could compute if there exists a derivation which uses all nonterminals of CFG in deterministic polynomial time, then we would be able to solve 3-SAT in this time, which is proved to be impossible. The conclusion is that described problem is NP-complete.